Теорія тестування для розробників

Укладач: Дмитро Попов.
https://www.linkedin.com/in/dima1popov

—

Пора нарешті розв'язати цей гордіїв вузол, що стосується TDD та автоматичного тестування фронтенду загалом.

Давайте я вам розповім, як це працює на практиці. Ви приходите на проект і хочете зробити все правильно: писати тести, бо наслухалися від різних спеціалістів про важливість тестування. Ви створюєте компонент і для нього пишете юніт-тести, мокаєте дещо, дещо — ні. Можливо, підключаєте проксі-сервери, щоб мокати запити. Але, зрештою, все це виглядає хаотично і не має системного підходу.

Перш за все, коли ви працюєте над фронтендом, у вас повинна бути бібліотека компонентів, або, іншими словами, дизайн-система. Якщо її немає — це набагато більша проблема, ніж відсутність тестів. Без цієї основи ваша робота з тестуванням стає малоефективною і обмеженою. Тому, якщо бібліотеки UI-компонентів ще не існує, або ви не використовуєте готову, вам слід створити свою. Всі компоненти в цій бібліотеці повинні бути ретельно протестовані за допомогою юніт-тестів і обов'язково задокументовані. Для цього можна використовувати інструменти на кшталт Storybook.

Зазначу важливий момент: якщо ви відмовляєтесь від тестування в проекті взагалі — цей текст не для вас, ви складний випадок. Якщо ж ви хочете рухатися вперед і писати якісні тести, потрібно починати з правильних основ.

Коли у вас буде бібліотека компонентів, покрита тестами, тоді можна говорити про розробку інших частин вашого проекту. Далі, насправді, вам залишатиметься лише писати юніт-тести для специфічних компонентів, таких як форми або модальні вікна, які ви створюєте, комбінуючи ваші UI-компоненти.

Важливо зазначити, що в компонентах не повинно бути прямих запитів до серверу, таких як "fetch" чи подібні. Всі ці запити мають мокатися та правильно передаватися в компонент, через props чи store. Але забутьте про використання проксі-серверів для тестування! Це хак. Запити або мокаються повністю, без проксі, або мають виконуватись в ізольованому середовищі, наприклад, у Docker-контейнері, хоча це вже більше для E2E тестів.

Далі вам залишиться покрити основні юзер-сценарії E2E (End-to-End) тестами, і на цьому більшість вашої роботи з тестуванням можна вважати завершеною. Звісно, є ще не функціональні тести, але це вже інша тема.

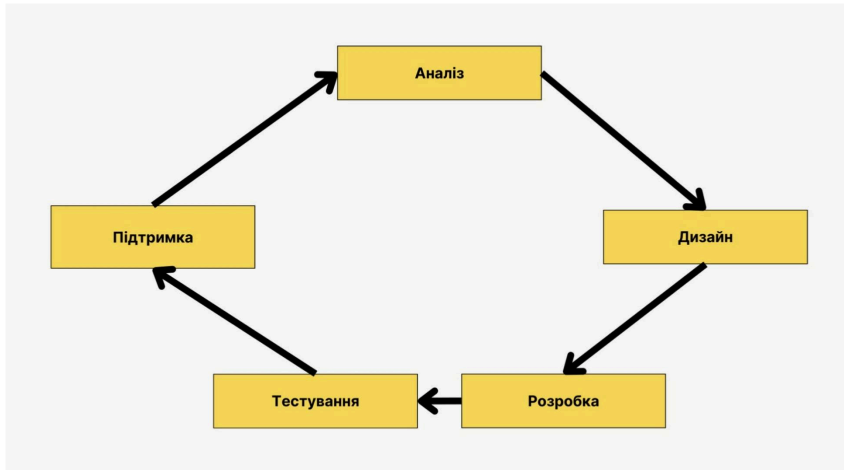
Щодо TDD в контексті фронтенду — вам просто не буде необхідності використовувати цей підхід у всіх випадках. Ви будете застосовувати його лише для тих компонентів, де це справді необхідно. Наприклад, коли ви працюєте з інпутом, який має складний регулярний вираз. В такому випадку для кожної зміни вам може знадобитися перевіряти десятки, а то й сотні різних комбінацій введених даних. І тут TDD може стати вашим рятівним інструментом, оскільки дозволить уникнути численних регресійних тестів, автоматизуючи їх. Або якщо ви пишете якийсь набір функцій типу свій lodash, тоді TDD також в самий раз. Для тестування використовуємо React Testing library, Jest, Playwright.

—

Тестів є багато, наприклад, тест Тюрінга, CAPTCHA, Тест на простоту числа, Тест на наявність вірусу, IQ-тест та інші. Теорія тестування дуже важлива частина інженерії.

“Все, що може піти не так, піде не так” (Закон Мерфі)
Закон Мерфі в контексті інженерії акцентує на важливості передбачення можливих проблем та помилок, які можуть виникнути в процесі розробки та експлуатації систем. Це спонукає інженерів до ретельного планування і тестування, особливо через негативні тести, які перевіряють систему на реагування на неправильні чи непередбачувані ситуації. Крім того, підхід “захисту від дурня” полягає в створенні систем, що здатні витримати неправильне використання або помилки користувачів, забезпечуючи безпеку, коректність введених даних та стійкість до збоїв. Обидва ці підходи сприяють створенню надійних, безпечних і стійких до помилок систем.
Якщо користувач вводить некоректні дані, система має вивести попередження або повідомлення, яке чітко вказує на проблему. Наприклад, якщо введений пароль не відповідає вимогам безпеки, система має попередити користувача з поясненням, що потрібно змінити (наприклад, "Пароль має містити принаймні 8 символів і одну велику літеру"). Це дозволяє уникнути подальших помилок і забезпечує правильну взаємодію з користувачем, що знижує ризик системних помилок через неправильний ввід.

Життєвий цикл програмного забезпечення - Systems (software) development life cycle (SDLC).



Цикл розробки програмного забезпечення (SDLC) включає такі етапи:
1. Збір та аналіз вимог,
2. Проєктування системи (дизайн),
3. Розробка (програмування),
4. Тестування,
5. Розгортання (deploy),
6. Підтримка та покращення системи (програми).

Кожен етап важливий для успішного завершення проєкту.
Зокрема, SDLC описаний у SWEBOK від IEEE.

“**Забезпечення якості програмного забезпечення** (Quality assurance) не є тестуванням. Забезпечення якості програмного забезпечення (Software Quality Assurance, SQA) — це набір заходів, які визначають і оцінюють адекватність процесів програмного забезпечення, щоб надати докази, які встановлюють впевненість у тому, що процеси програмного забезпечення є відповідними та створюють програмні продукти відповідної якості для їх цільового призначення. Ключовим атрибутом SQA є об’єктивність функції SQA щодо проекту” (SWEBOK, IEEE).

Тестування — процес виконання тестів.
Тестування має метод, тестуємий об'єкт, результат.

Мета тестування: Збільшення ймовірності того, що система відповідає вимогам.

Тестування показує наявність дефектів, а не їх відсутність. Варто казати, що помилок не було виявлено, а не що їх немає.

Почніть з припущення, що програма містить помилки (що є актуальним припущенням для майже будь-якої програми), а потім тестуйте програму, щоб знайти якомога більше помилок.
Таким чином, більш доречне визначення звучить так:
Тестування — це процес виконання програми з метою виявлення помилок.

Тест — один або кілька тестових випадків (test case).

Тестовий випадок — це набір передумов, введених даних, дій, очікуваних результатів, розроблених на основі умов тестування.

Позитивний тест (результат) - тест, який повідомляє про помилку, тобто failed тест.

Хибно-позитивний результат тесту (false-positive result) - результат, який свідчить про наявність дефекту, якого насправді немає.

У процесі тестування програмного забезпечення тестуються як функціональні, так і нефункціональні вимоги.

Функціональні вимоги визначають, що система повинна робити, тоді як нефункціональні вимоги визначають якість або характеристики, які повинні бути притаманні системі, такі як продуктивність, надійність, безпека тощо.
Тестування може бути за принципом чорної, чи білої скриньки.

Годинник повинен виконувати такі функціональні вимоги: відображати поточний час, дату, підтримувати будильник, секундомір, час для кількох часових поясів та зміну формату часу (12/24 години). До нефункціональних вимог належать: висока точність (±1 секунда на добу), енергоефективність (робота батареї не менше року), водонепроникність (до 50 метрів), міцність (витримування падіння з 1 метра), зручний інтерфейс, сучасний дизайн, швидкий відгук на команди (до 0,5 секунди) та синхронізація з мобільними пристроями через Bluetooth.

Однією з важливих стратегій тестування є тестування за принципом чорної скриньки (також відоме як тестування на основі даних або на основі вводу/виводу). Щоб скористатися цим методом, розгляньте програму як чорний ящик, який щось отримує і щось видає, та про механізм якого вам не відомо.
У тестуванні за принципом чорної скриньки тестові дані отримують виключно зі специфікацій (тобто без використання знання внутрішньої структури програми). Інша стратегія тестування, білий ящик (або логічно кероване тестування), дозволяє перевірити внутрішню структуру програми. Ця стратегія отримує тестові дані з перевірки логіки програми (і часто, на жаль, без урахування специфікації).

Тестування за принципом чорної скриньки – це підхід, за якого тестувальник не має доступу до внутрішньої структури чи коду системи. Тестування базується на аналізі вимог та очікуваних результатів.
Це схоже на роботу з пристроєм, внутрішній механізм якого невідомий: тестувальник бачить лише результат роботи.

Приклад: Тестування калькулятора, де вводять числа і оператори (наприклад, 2 + 2) та перевіряють, чи результат дорівнює 4, не знаючи, як працює внутрішня логіка.

Взагалі багато людей живе у світі чорних скриньок, однак інженер повинен жити у світі білих скриньок, тобто розуміти як працюють прилади.
"Моїм незмінним питанням при отриманні будь-якої нової іграшки було: Мамо, а що всередині?" (Чарльз Беббідж, "Уривки з життя філософа")
Тестування за принципом чорної скриньки потрібно інженеру методологічно, бо допомагає сфокусуватися на вимогах, а не на коді (системі).

Тестування за принципом білої скриньки – це підхід, за якого тестувальник має повний доступ до внутрішнього коду, алгоритмів та структури системи. Тут перевіряються не лише результати роботи системи, але й правильність реалізації коду, покриття тестами умов, циклів чи шляхів виконання програми.

Приклад: Аналіз коду функції для перевірки, чи всі можливі умови (наприклад, if та else) опрацьовуються належним чином.

Приклад тесту. Тест на простоту числа.

Мета: Перевірити функцію, яка визначає, чи є число простим.

Передумови:
1. Є функція isPrime(num), яка приймає ціле число як аргумент.
2. Просте число визначається як число більше 1, яке ділиться лише на 1 і на себе.

Тестові випадки:

1. Тестовий кейс (випадок) 1:
   - Вхідні дані: 2
   - Очікуваний результат: true (2 — просте число)

2. Тестовий кейс 2:
   - Вхідні дані: 4
   - Очікуваний результат: false (4 — не просте число)

Реалізація тесту на JavaScript:

```
function isPrime(num) {
  if (num <= 1) return false;
  for (let i = 2; i <= Math.sqrt(num); i++) {
    if (num % i === 0) return false;
  }
  return true;
}

// Тестові випадки
console.log(isPrime(2));  // Очікуємо true
console.log(isPrime(4));  // Очікуємо false
console.log(isPrime(17)); // Очікуємо true
console.log(isPrime(9));  // Очікуємо false
```

Дефект — певна невідповідність системи її специфікації.
Баг - це, зокрема, синтаксична, або логічна помилка.
Усі баги є дефектами. Протилежне не завжди вірно, хіба ви використовуєте баг і дефект як синоніми. Баг можна розглядати як причину дефекту.
Логічна помилка виникає, коли логіка у коді є хибною, навіть якщо синтаксис правильний. Такі помилки часто призводять до непередбачуваної поведінки.
Синтаксичні помилки виникають, коли код не відповідає правилам синтаксису мови програмування. Такі помилки зазвичай виявляються на етапі компіляції (для компільованих мов) або під час виконання (для інтерпретованих мов).

В 60х роках, в словнику "Electronics and nucleonics dictionary" (1960), автор John Markus, пише: "**баг -** сленговий термін для позначення несправності в частині обладнання". В словнику також згадано про дебагінг на основі breakpoints. Програма дебагер (debugger) з крокуванням по breakpoints використовувалася в IBM в 1960-х роках.

Термін "баг" (bug) у контексті комп'ютерних помилок асоціюється з Грейс Хоппер, американською комп'ютерною науковицею і адміралом ВМС США. У 1947 році, коли вона працювала над комп'ютером Mark II в Гарвардському університеті, її команда виявила міль, яка потрапила в одну з машин і спричинила несправність. Вони жартівливо записали в журналі, що знайшли "баг" (bug) — це стало відомим прикладом використання терміну "баг" для опису помилок у програмному забезпеченні. Проте в той час, словом баг називали прилади для прослуховування в електроніці.

Грейс Хоппер також зробила значний внесок у розробку компіляторів і була однією з розробників мови програмування COBOL, активно підтримуючи ідею автоматичного налагодження помилок (зневадження) у програмному коді.

Фейл, несправність, провал (failure) – це подія, за якої компонент або система не відповідає своїм вимогам у межах визначених обмежень.

Mistake (помилка, похибка) – це помилка спричинена людським фактором. **Error** – ширше поняття, є HTTP errors, Compiler error s, JS errors.

Некоректні результати не завжди є результатом багів чи дефектів в системі, вони можуть бути спричинені природою системи, наприклад, якщо система робить передбачення, працює на основі ймовірнісних чи евристичних алгоритмів. Для таких систем вводять поняття **акуратність**.

Акуратність моделі (точність моделі) — це відношення загальної кількості відповідей, до кількості коректних відповідей. Наприклад, серед 100 відповідей 10 коректних, тоді акуратність буде 10/100;

Severity та Priority — це два важливі поняття в тестуванні програмного забезпечення, які використовуються для класифікації дефектів або помилок.

1. Severity (Серйозність, суворість)
Severity описує технічний вплив помилки на систему. Це вказує на те, наскільки серйозною є проблема для функціонування програми. Інакше кажучи, це характеристика того, наскільки сильно помилка впливає на програму чи її компонент.

Типи Severity:
- Critical: Помилка блокує роботу програми або призводить до її аварійного завершення (наприклад, програма не запускається).
- Major: Важка помилка, яка суттєво обмежує функціональність, але не призводить до аварії (наприклад, відсутність функції для збереження даних).
- Minor: Легка помилка, яка не сильно впливає на загальну роботу програми (наприклад, некоректне відображення тексту).
- Trivial: Незначна помилка, яка не має суттєвого впливу на функціональність (наприклад, орфографічна помилка в повідомленні).

2. Priority (Пріоритет)
Priority вказує на терміновість виправлення помилки. Це описує, як швидко потрібно виправити проблему, базуючись на її важливості для користувачів або бізнес-процесів. Вона визначається з огляду на вплив на бізнес чи потреби клієнтів.

Типи Priority:
- High: Виправлення помилки повинно бути здійснено якомога швидше, оскільки проблема серйозно впливає на користувачів чи бізнес.
- Medium: Помилка повинна бути виправлена у межах стандартного терміну, тому що вона не критична, але має вплив.

- Low: Помилка може бути виправлена пізніше, оскільки її вплив мінімальний.

Severity стосується того, наскільки серйозна помилка з технічної точки зору.
Priority стосується того, як швидко потрібно виправити помилку на основі її впливу на користувачів або бізнес.

Баг-репорт (bug report) — це документ, який описує знайдену помилку (баг) у програмному забезпеченні. Його основна мета — повідомити розробників про проблему, щоб вони могли її виправити. Правильно складений баг-репорт допомагає зекономити час і зусилля команди розробки.

Структура баг-репорту
1. Заголовок (Summary)
Короткий опис проблеми. Має бути чітким і конкретним, наприклад:
"Не працює кнопка 'Зберегти' на формі створення користувача"

2. ID або номер
Унікальний ідентифікатор багу (присвоюється автоматично в системах трекінгу, таких як Jira, Trello, Bugzilla).

3. Кроки для відтворення (Steps to Reproduce)
Покрокова інструкція, яка дозволяє повторити помилку. Наприклад:
   1. Відкрити сторінку створення користувача.
   2. Заповнити всі поля.
   3. Натиснути кнопку "Зберегти".

4. Очікуваний результат (Expected Result)
Опис того, як має працювати функціонал за нормальних умов.
"Дані користувача мають зберегтися, і з'явиться повідомлення 'Успішно збережено'."

5. Фактичний результат (Actual Result)
Реальний результат виконання дій.
"Кнопка не реагує, дані не зберігаються."

6. Оточення (Environment)
Деталі про середовище, де виникла помилка:
   - Версія операційної системи.
   - Браузер (якщо це веб-додаток).
   - Версія додатка або build.
   - Пристрій (телефон, планшет, ПК).

7. Пріоритет і серйозність (Priority and Severity)
   - Пріоритет: наскільки важливо виправити баг (визначає менеджер/product owner).

   - Серйозність: рівень впливу на функціонал (критичний, високий, середній, низький).

8. Скріншоти або відео
Візуальне підтвердження багу для полегшення діагностики.

9. Логи або трейсинг (Log Files/Stack Trace)
Логи помилок, які допомагають розробникам зрозуміти технічні причини.

Основні принципи створення баг-репорту
1. Точність: Чітко описуйте проблему, без двозначностей.
2. Короткість: Уникайте зайвих деталей.
3. Повнота: Вкажіть усю необхідну інформацію для відтворення проблеми.
4. Об'єктивність: Уникайте емоцій або припущень.

Приклад баг-репорту
- Заголовок: Помилка при натисканні кнопки 'Додати товар' у браузері Safari
- Кроки:
   1. Відкрити сайт у Safari (версія 15.0).
   2. Увійти в систему.
   3. Спробувати додати товар у кошик.
- Очікуваний результат: Товар додається до кошика.
- Фактичний результат: Нічого не відбувається, у консолі з’являється помилка `TypeError: undefined`.
- Оточення: macOS 12.0.1, Safari 15.0.
- Пріоритет: Високий.
- Скріншоти: Додаються.

Такий підхід до створення баг-репорту забезпечує ефективну комунікацію між тестувальниками та розробниками.

Верифікації та валідації

“Верифікація — це спроба забезпечити правильність створення продукту, тобто відповідність результатів діяльності специфікаціям, визначеним на попередніх етапах.
Валідація — це спроба переконатися, що створено саме той продукт, який потрібен, тобто продукт виконує своє конкретне призначення.
Процеси верифікації та валідації починаються ще на ранніх етапах розробки або підтримки продукту. Вони забезпечують перевірку ключових характеристик продукту стосовно його безпосереднього попередника та відповідності заданим специфікаціям” (SWEBOK, IEEE).

Процеси верифікації та валідації проводяться за допомогою різних методів і підходів залежно від стадії розробки та типу продукту. Ось основні кроки для кожного процесу:

Верифікація (Verification)

Мета: Переконатися, що продукт створений правильно відповідно до специфікацій.
1. Огляд документації:
   - Аналіз технічних вимог, дизайну, архітектури, тестових планів.
   - Перевірка узгодженості між документами.
2. Статичне тестування:
   - Рев'ю коду (code review).
   - Аналіз статичними інструментами, такими як лінтери чи статичні аналізатори.
3. Аналіз:
   - Перевірка на відповідність стандартам, правилам і специфікаціям.
   - Використання чек-листів для перевірки.
4. Моделювання та симуляція:
   - Перевірка компонентів системи через симуляцію їх роботи до фізичного тестування.

Валідація (Validation)
Мета: Переконатися, що створено правильний продукт, який відповідає потребам користувачів.
1. Динамічне тестування:
   - Виконання функціональних, інтеграційних, системних і приймальних тестів.
   - Перевірка роботи продукту в реальних або наближених до реальних умовах.
2. Користувацьке тестування:
   - Проведення тестів за участі кінцевих користувачів (UAT, User Acceptance Testing).
   - Оцінка зручності використання (usability testing).
3. Прототипування:
   - Створення прототипів або MVP (minimum viable product) для перевірки концепції.
4. Пілотний запуск:
   - Запуск продукту в обмеженому середовищі для збору відгуків і виявлення проблем.
5. Оцінка відповідності:
   - Перевірка, чи відповідає продукт бізнес-цілям, потребам ринку та користувачів.

Загальні методи для обох процесів
- Використання тестових сценаріїв (test cases) для різних рівнів тестування.
- Автоматизація тестів для покращення точності та ефективності.
- Регулярні рев'ю ітерацій продукту в межах Agile чи інших методологій.

Обидва процеси повинні повторюватися на кожному етапі життєвого циклу розробки продукту, щоб забезпечити його якість і відповідність вимогам.

Автентифікація та авторизація — це два важливі процеси в системах безпеки, які часто використовуються разом, але мають різні функції:

1. Автентифікація (Authentication) — це процес перевірки особи користувача, тобто перевірка, чи є ви тим, ким ви стверджуєтеся. Це зазвичай здійснюється за допомогою таких методів, як:
   - Пароль

- Біометричні дані (відбитки пальців, розпізнавання обличчя тощо)
- Двофакторна аутентифікація (наприклад, поєднання пароля та одноразового коду)

Приклад: Ви вводите логін і пароль на вебсайті, і система перевіряє, чи є ці дані правильними.

2. Авторизація (Authorization) — це процес надання або обмеження доступу до певних ресурсів або операцій після того, як користувач успішно пройшов автентифікацію. Авторизація визначає, що користувач може або не може робити в системі, наприклад:
   - Які сторінки чи функції доступні користувачу
   - Які операції він може виконувати (читання, редагування, видалення)

Приклад: Після того, як ви ввійшли в систему (автентифікація), система визначає, чи маєте ви права редагувати документи або тільки переглядати їх (авторизація).

Ключова різниця:
- Автентифікація перевіряє хто ви.
- Авторизація визначає, що ви можете робити після того, як система підтвердить вашу особу.

7 принципів тестування (згідно з ISTQB)

Один з найважливіших елементів розробки програмного забезпечення — це тестування. Це може бути надзвичайно складно організувати правильно, щоб забезпечити максимальну ефективність. Тому завжди варто переглядати свої процеси та методологію, щоб переконатися, що дотримуються найкращі практики.

Гарною відправною точкою є список 7 принципів тестування Міжнародної ради з кваліфікацій тестування програмного забезпечення (ISTQB).

7 принципів тестування від ISTQB.

1. Тестування показує наявність дефектів, а не їх відсутність.

Метою тестувальної команди повинно бути підтвердження того, що продукти чи застосунки, що тестуються, можуть працювати та відповідати потребам кінцевого користувача, а також виконувати бізнес-вимоги. Завдання тестувальників — не довести, що продукт без дефектів або помилок, оскільки це майже неможливо. Замість цього використовуються різні типи тестування та методології, щоб знайти та виявити приховані дефекти.

2. Вичерпне (exhaustive) тестування неможливе.

Саме тому, дизайн тестів необхідний, щоб зробити тестування якомога повним (complete). Вичерпне тестування вимагало б нескінченної кількості зусиль, що, безумовно, було б надзвичайно неефективним та перевищило б терміни проекту. Натомість пріоритет надається різним технікам тестування для конкретних комбінацій, щоб забезпечити оптимальний рівень тестування на основі оцінки ризику того, що тестується.

3. Раннє тестування економить час і гроші.

Тестування повинно починатися на найраннішому етапі, щоб виявити дефекти чи помилки якомога раніше. Швидкий старт дозволяє обмежити кількість проблем, виявлених на пізніших етапах тестування, що економить гроші. Як правило, тестування можна почати після визначення вимог до тесту.

“Унція профілактики коштує фунта лікування”, “Час — це гроші” — Бенджамін Франклін, 1748 рік.

4. Дефекти збираються разом.
Під час тестування більшість виявлених дефектів можуть бути пов'язані з кількома модулями, які є особливо складними або містять застарілий код. Це принцип, схожий на принцип Парето: 80% дефектів можна знайти в 20% модулів, що тестуються. Знання про ці модулі допомагає тестувальникам зосередити зусилля на важливих ділянках.

5. Остерігайтеся парадоксу пестицидів.
Цей принцип стверджує, що якщо виконувати одні й ті ж тести повторно, вони з часом не зможуть виявити нові дефекти. Це схоже на те, як шкідники з часом розвивають імунітет до пестицидів. Тестувальники повинні постійно переглядати та змінювати свої тест-кейси для максимізації ефективності.

6. Тестування залежить від контексту.
Тестування залежить від контексту, оскільки для різних типів програм необхідний специфічний підхід. Наприклад, програма для круїзної індустрії буде суттєво відрізнятися від програми для страхової компанії.

7. Відсутність помилок — це хибне уявлення.
Програмне забезпечення, яке було протестоване та виявилося на 99% без помилок, може все одно бути непрацездатним, якщо система тестувалася за неправильними вимогами. Тестування не лише виявляє дефекти, але й оцінює, чи відповідає програмне забезпечення вимогам кінцевих користувачів.

Принципи тестування Гленфорда Маєрса, з книги “Мистецтво тестування програмного забезпечення”:

1. Необхідною частиною тест-кейсу є визначення очікуваного результату.

2. Програміст не повинен намагатися тестувати свою програму виключно самостійно.

Будь-який письменник знає — або повинен знати — що це погана ідея — намагатися редагувати чи вичитувати свою власну роботу. Вони знають, що має сказати твір, тому можуть не розпізнати, коли в ньому сказано інше. І вони дуже не хочуть знаходити помилки у власній роботі. Те саме стосується і авторів програмного забезпечення. Ще одна проблема виникає зі зміною фокусу на програмному проекті. Після того, як програміст конструктивно спроектував і закодував програму, надзвичайно важко раптово змінити перспективу, щоб поглянути на програму деструктивним поглядом.

3. Програмістська організація не повинна тестувати власні програми тільки самостійно.

4. Будь-який процес тестування має включати ретельну перевірку результатів кожного тесту.

5. Тест-кейси повинні бути написані для некоректних і несподіваних умов, а також для коректних і очікуваних.

6. Тестування повинно перевіряти не тільки те, що програма не робить того, що повинна, а й те, що вона робить, чого не повинна.

7. Уникайте “викидних” тест-кейсів, якщо програма не є “викидною”.

Цю проблему найчастіше можна побачити в інтерактивних системах для тестування програм. Загальною практикою є сидіти за терміналом, вигадувати тестові випадки на ходу і потім відправляти ці тестові випадки через програму. Основною проблемою є те, що тестові випадки являють собою цінне інвестиційне зусилля, яке в такому середовищі зникає після завершення тестування. Кожного разу, коли програму потрібно тестувати знову (наприклад, після виправлення помилки або покращення), тестові випадки доводиться вигадувати заново. Чим частіше це трапляється, тим більше, оскільки цей процес вимагає значних зусиль, люди схильні уникати його.

Тому повторне тестування програми рідко буває таким же суворим, як первинне, що означає, що якщо зміна спричинить збій раніше працездатної частини програми, ця помилка часто залишається непоміченою. Збереження тестових випадків і їх повторний запуск після змін в інших компонентах програми відоме як **регресійне тестування**.

8. Не плануйте тестування, припускаючи, що помилок не буде знайдено.

9. Ймовірність виявлення помилок пропорційна кількості вже знайдених помилок у програмі.

10. Тестування є надзвичайно творчим і інтелектуально складним завданням.

Принципи, які ми вважаємо важливими для гнучкого тестувальника (from Lisa Crispin' book "Agile testing"):

- Надавайте постійний зворотний зв'язок.
- Забезпечуйте цінність для клієнта.
- Сприяйте спілкуванню віч-на-віч.
- Будьте сміливими.
- Дотримуйтеся простоти.
- Практикуйте постійне вдосконалення.
- Реагуйте на зміни.
- Самоорганізуйтеся.
- Зосереджуйтеся на людях.
- Насолоджуйтеся процесом.

Презумпція невинуватості щодо програміста в контексті тестування також може бути інтерпретована як підхід, за якого програміст не вважається винним у дефектах чи помилках, поки ці помилки не будуть виявлені під час тестування чи перевірки. Це означає, що до тестування програмісти повинні вважатися "невинними" у контексті роботи коду, а сам процес тестування має на меті знайти помилки або недоліки, а не звинувачувати програміста.

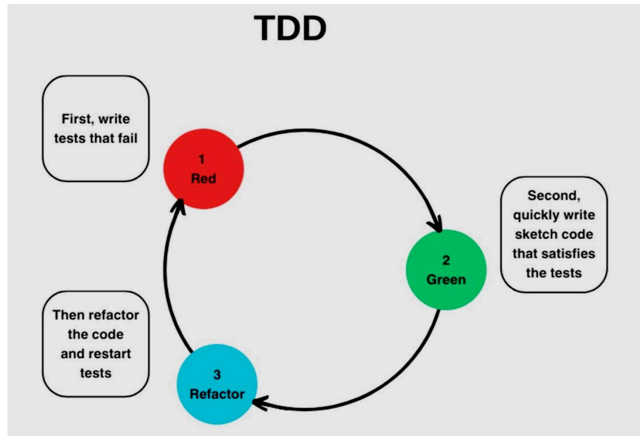
У цьому випадку важливо, щоб тестувальники і менеджери проектів не поспішали оцінювати програміста як винного в помилках, а зосереджувалися на тому, щоб виявити проблеми, з'ясувати причини їх виникнення і вирішити їх. Це також означає, що програмістам надається можливість виправити помилки, не відчуваючи постійного тиску чи недовіри.

З точки зору практики, це сприяє здоровій атмосфері у команді, де програміст може працювати без страху бути звинуваченим за кожну дрібну помилку, і тестувальники можуть зосереджуватися на своїй роботі, не припускаючи винуватості на ранніх етапах.

TDD

TDD (Test-Driven Development) — це метод розробки програмного забезпечення, при якому спочатку пишуться тести, а потім код, що ці тести проходить.

Кент Бек у своїй відомій книзі про TDD (Керована тестами розробка) описує TDD мантру.



TDD (Test-driven development) мантра Червоний/зелений/рефакторинг — мантра TDD.

1. Червоний — Напишіть невеликий тест, який не працює, і, можливо, спочатку навіть не компілюється.

2. Зелений — Змусити тест працювати швидко, не витрачаючи час на доведення коду до ідеалу.

3. Рефакторинг — Усуньте всі дублювання, створені просто для того, щоб змусити тест працювати.

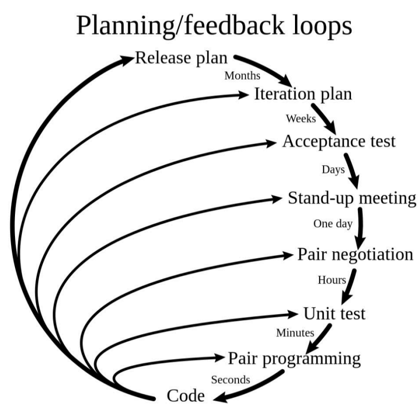
Принципи TDD від Боба Мартіна (Clean code).

Перший закон: ви не можете писати робочий код, доки не напишете невдалий модульний тест (юніт тест).

Другий закон: ви не можете написати більше модульного тесту, ніж достатньо для невдачі, а відсутність компіляції означає невдачу.

Третій закон: ви не можете писати більше робочого коду, ніж достатньо для проходження поточного невдалого тесту.

Цикл розробки в Extreme programming на основі TDD та Agile.



BDD

На рубежі тисячоліть Ден Норт почав працювати над перевизначенням TDD, яке він назвав Behavior-Driven Development (BDD). Його мета полягала в тому, щоб усунути технічний жаргон (глосарій) із тестів і зробити тести більш схожими на специфікації, які оцінили б бізнесмени.

BDD (Behavior-Driven Development) — це методологія розробки програмного забезпечення, яка акцентує увагу на поведінці системи з точки зору бізнесу та користувача. BDD поєднує автоматизоване тестування, розробку, та бізнес-аналіз, забезпечуючи краще розуміння очікувань системи всіма учасниками процесу.

BDD зазвичай використовує специфічну структуру для написання тестових сценаріїв:

Приклад:
Given я ввів число 2 у калькулятор
And я ввів число 3
When я натискаю кнопку "Додати"
Then результат має бути 5.

Розділи:
- Given (Передумова): Стан системи до дії.
- When (Дія): Що робить користувач або система.
- Then (Очікуваний результат): Поведінка системи після виконання дії.

BDD зазвичай інтегрується через спеціальні бібліотеки, такі як Cucumber.

Як це працює:
1. Тестові сценарії пишуться мовою Gherkin.

2. Ці сценарії зв’язуються з кодом тестів через "step definitions" — реалізації дій, описаних у сценаріях.

Приклад коду:
Gherkin-сценарій:

```
Scenario: Додавання чисел
  Given я ввів число 2
  And я ввів число 3
  When я натискаю "Додати"
  Then результат має бути 5
```

Step Definition (JavaScript з Cucumber):

```
Given('я ввів число {int}', function (number) {
  this.calculator.enterNumber(number);
});

When('я натискаю {string}', function (button) {
  this.calculator.pressButton(button);
});

Then('результат має бути {int}', function (expectedResult) {
  assert.equal(this.calculator.result, expectedResult);
});
```

BDD будується на основі ідей TDD, але з акцентом на опис поведінки, а не окремих методів.
BDD часто використовується в Agile-проектах, де команди активно співпрацюють із клієнтами.

BDD допомагає сфокусуватися на тому, як система має працювати для користувачів, і забезпечує чітке спілкування між усіма учасниками проекту.

Давайте розглянемо цей приклад для тестування:

```
class Greeter {
  sayHello() {
    return 'hello'
  }
```

```
}

module.exports = {
  Greeter
}
```

```
Feature: Greeting

  Scenario: Say hello
    When the greeter says hello
    Then I should have heard "hello"
```

```
const assert = require('assert')
const { When, Then } = require('@cucumber/cucumber')
const { Greeter } = require('../../src')

When('the greeter says hello', function () {
  this.whatIHeard = new Greeter().sayHello()
});

Then('I should have heard {string}', function (expectedResponse) {
  assert.equal(this.whatIHeard, expectedResponse)
});
```

Проектування за контрактом (англ. Design by Contract, DbC) — це методологія розробки програмного забезпечення, запропонована Бертраном Мейєром, яка передбачає створення формальних угод (контрактів) між компонентами системи. Ці контракти описують обов'язки, гарантії та обмеження для обох сторін: постачальника (того, хто надає функціонал) і споживача (того, хто використовує цей функціонал).

На відміну від TDD, тести в DbC пишуться під час написання коду та є інтегрованими в сам код, зокрема, вони виконуються під час виконання програми.

Основні елементи проектування за контрактом:

1. Передумови (Preconditions)
   - Умови, які мають бути виконані перед викликом методу чи функції.
   - Це відповідальність споживача функціоналу.

- Приклад: Перед тим як зняти гроші з рахунку, сума зняття має бути меншою або рівною залишку на рахунку.

2. Післяумови (Postconditions)
   - Умови, які мають бути виконані після виконання методу чи функції, якщо передумови були дотримані.
   - Це гарантії, які надає постачальник.
   - Приклад: Після зняття грошей залишок на рахунку має зменшитися на відповідну суму.

3. Інваріанти (Invariants)
   - Умови, які мають завжди залишатися істинними протягом існування об’єкта чи системи (крім моментів зміни стану).
   - Приклад: Залишок на банківському рахунку ніколи не може бути від’ємним.

Переваги проектування за контрактом:

- Надійність: Чіткі контракти запобігають неправильному використанню компонентів і допомагають швидше знаходити помилки.
- Самодокументування коду: Контракти пояснюють, як має працювати компонент, без додаткової документації.
- Інкапсуляція Компоненти чітко визначають свої залежності та обов’язки, що покращує модульність.
- Легкість підтримки: Розробникам легше розуміти поведінку коду без необхідності глибокого аналізу його реалізації.

Реалізація

Хоча Eiffel підтримує DbC "із коробки", у багатьох інших мовах програмування його можна реалізувати через асерти або спеціальні бібліотеки.

Приклад у Python:

```
class BankAccount:
    def __init__(self, balance):
        assert balance >= 0, "Початковий баланс має бути не від’ємним"
        self.balance = balance

    def withdraw(self, amount):
        # Передумова
        assert amount > 0, "Сума зняття має бути позитивною"
        assert amount <= self.balance, "Недостатньо коштів на рахунку"

        # Основна логіка
        self.balance -= amount
```

```
# Післяумова
assert self.balance >= 0, "Баланс не може бути від’ємним"
return self.balance
```

Реальні застосування

- Розробка API: Контракти допомагають чітко визначити допустимі вхідні/вихідні дані та очікувану поведінку.
- Критичні системи: Використовується там, де помилка є недопустимою (медичні пристрої, авіація).
- Тестування: Контракти слугують вбудованим механізмом перевірки умов під час виконання коду.

Введення асертів і перевірок на кожному кроці роботи програми може призвести до затримок, особливо в масштабних системах, де кожен додатковий обчислювальний крок має значення.
Design by Contract — це потужний підхід до забезпечення коректності коду, але його використання може бути обмежене в середовищах, де важлива швидка адаптація, гнучкість та продуктивність. У великих системах, де часто змінюються вимоги або де важлива масштабованість, DbC може стати важким у впровадженні і підтримці.
Бертран Мейєр створив мову програмування Eiffel та концепцію проєктування за контрактом, а також принцип відкритості/закритості, що серед принципів SOLID.

Exploratory testing

Експлоративне тестування — це стиль тестування, який акцентує увагу на швидкому циклі навчання, проєктування тестів і виконання тестів. Замість того, щоб намагатися перевірити відповідність програмного забезпечення заздалегідь написаному сценарію тестування, експлоративне тестування досліджує характеристики програмного забезпечення, виявляючи аспекти, які можуть бути класифіковані як очікувана поведінка або як збої. Сценарне тестування може перевірити лише те, що передбачено у сценарії, фіксуючи лише відомі умови. Такі тести можуть бути надійною сіткою, яка ловить помилки, але як дізнатися, чи охоплює ця сітка все, що необхідно?

Експлоративне тестування прагне перевірити межі цієї сітки, знаходячи нові поведінки, які не включені до жодного сценарію. Воно часто виявляє нові помилки, які можна додати до сценаріїв, а іноді виявляє поведінку, що є безпечною або навіть бажаною, але про неї раніше не думали. Експлоративне тестування є набагато гнучкішим та менш формальним процесом, ніж сценарне тестування, але все одно потребує дисципліни для ефективного виконання. Хороший спосіб організувати цей процес — проводити експлоративне тестування у сесії з обмеженим часом. Такі сесії зосереджуються на конкретному аспекті програмного забезпечення.

Експлоративне тестування передбачає випробування різних підходів, отримання знань про те, як працює програмне забезпечення, застосування цих знань для формування питань і гіпотез, а також створення нових тестів у процесі, щоб зібрати більше інформації. Експлоративне тестування має бути регулярною діяльністю, що відбувається протягом усього процесу розробки програмного забезпечення. Деякі команди проводять експлоративне тестування протягом приблизно пів години після завершення роботи над історією. Мартін Фаулер вважає це тривожним сигналом, якщо команда взагалі не проводить експлоративного тестування — навіть якщо їх автоматизовані тести чудові. Навіть найкраще автоматизоване тестування є за своєю природою сценарним, і цього недостатньо.

На практиці експлоративне тестування починається з визначення мети: тестувальник обирає конкретну частину продукту (наприклад, форму входу) і обмежує час сесії (30–60 хвилин). У ході тестування досліджуються різні сценарії використання, включаючи нестандартні чи екстремальні дії: введення некоректних даних, перевірка роботи під час повільного інтернету або одночасне натискання кнопок. Усі дії та їхні результати ретельно фіксуються для подальшого аналізу.

Після завершення тестування результати класифікуються: виявлені помилки документуються, цікаві або нові сценарії додаються до майбутніх тестів. Такий підхід допомагає знайти поведінку, яка не була передбачена в автоматизованих чи сценарних тестах, і забезпечити глибше розуміння продукту. Регулярне впровадження експлоративного тестування, наприклад, після завершення кожної історії, допомагає підтримувати якість розробки на високому рівні.

User story (користувацька історія) — це короткий опис функціональності або вимоги до системи з точки зору користувача або замовника. Зазвичай це невеликий фрагмент роботи, який може бути реалізований за один цикл розробки. User story допомагає визначити, що потрібно створити або вдосконалити в продукті, і має бути зрозумілою для всіх членів команди, щоб вони могли працювати над її реалізацією.

Зазвичай user story має наступну структуру:

Як [тип користувача], я хочу [щоб система виконувала певну функцію], щоб [отримати певну вигоду або результат].

Приклад: Як користувач, я хочу мати можливість змінювати пароль, щоб забезпечити безпеку свого облікового запису.

User stories використовуються в методології Agile для планування і пріоритизації завдань. Вони можуть мати додаткові критерії прийняття, щоб чітко визначити, що означає "готово".

Під час написання User story врахуйте принцип **KISS** - Keep It Simple, Stupid.

Вхідні та вихідні критерії тестування (Test entry and exit criteria)

Не пишіть більше модульного тесту, ніж достатньо для невдачі, достатньо для не проходження Definition of Done.

1. **Definition of Done** (DoD)
Definition of Done — це набір критеріїв, які визначають, коли завдання (наприклад, User Story або дефект) вважається завершеним і готовим до випуску.

Основні аспекти:
- Код написаний, перевірений (Code Review) і злитий у головну гілку.
- Виконані всі функціональні та нефункціональні тести (автоматичні, ручні).
- Усі критичні дефекти виправлені.
- Оновлена документація (технічна та користувацька).
- Схвалення власника продукту (Product Owner).

У контексті вихідних критеріїв тестування, DoD гарантує, що всі тести пройдені й система готова до випуску.

2. **Definition of Ready** (DoR)
Definition of Ready — це набір критеріїв, які визначають, коли завдання (User Story, епік або дефект) готове до роботи.

Основні аспекти:
- Завдання має чіткі, зрозумілі та перевірені вимоги.
- Є чіткі Acceptance Criteria.
- Завдання оцінено командою (наприклад, у story points).
- Необхідні залежності вирішені (наприклад, API доступне).
- Тестові дані або середовище готові.

У контексті вхідних критеріїв тестування, DoR допомагає переконатися, що перед початком роботи над тестуванням все підготовлено.

3. **Критерії INVEST**
Критерії INVEST використовуються для створення якісних User Stories.

- I – Independent (Незалежні): User Stories не повинні залежати одна від одної.
- N – Negotiable (Договірні): Можна змінювати та уточнювати вимоги до User Story.
- V – Valuable (Цінні): Завдання має приносити цінність користувачу.
- E – Estimable (Оцінювані): Завдання має бути достатньо зрозумілим для оцінки.
- S – Small (Малі): Завдання повинно бути розміром, який можна завершити за один спринт.

- T – Testable (Тестовані): Завдання має чіткі критерії прийняття, які можна перевірити тестами.

Принцип Testable прямо пов'язаний із підготовкою вхідних критеріїв тестування.

4. Estimation у контексті тестування
Оцінка (Estimation) — це процес визначення часу чи ресурсів, необхідних для виконання завдання.

У тестуванні оцінювання залежить від:
- Обсягу роботи: Скільки тест-кейсів потрібно створити та виконати.
- Рівня складності: Чи потрібне створення тестових даних або налаштування середовища.
- Ризиків і невизначеностей: Наприклад, можливі проблеми з інтеграцією.
- Автоматизації: Час, необхідний для написання автотестів.

Вхідні критерії тестування впливають на оцінку:
- Якщо середовище або тестові дані ще не готові, оцінка може бути неточною.

Вихідні критерії тестування впливають на завершення:
- Якщо не всі критерії виконані (наприклад, знайдені дефекти), етап тестування може бути продовжений, що впливає на початкові оцінки.

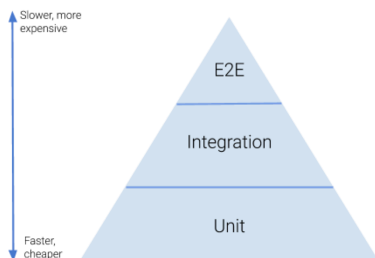
Методи оцінювання:
- Planning Poker.
- T-Shirt Sizing (Small, Medium, Large).
- Bucket System.

Оцінка безпосередньо залежить від відповідності DoR та DoD, оскільки вони задають чіткі рамки для виконання задач.

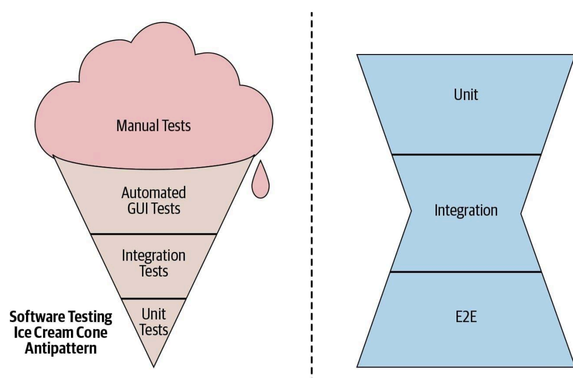
Хороша практика коли розробник та тестувальник дають одну загальну оцінку для завдання.

Рівні тестування

- Модульні тести (юніт тести).
- Інтеграційні тести.
- Системні тести.
    - E2E.
    - Приймальні тести.
    - Smoke-тестування.
    - Exploratory testing.



Команда тестування Google попереджає про два антипатерни: “ріжок морозива” та “пісочний годинник”. У випадку антипатерну “ріжок морозива” розробники пишуть занадто багато наскрізних (end-to-end) тестів і занадто мало інтеграційних та модульних тестів. Такі набори тестів, як правило, є повільними, ненадійними і складними у використанні.



У антипатерні “пісочний годинник” розробники пишуть багато наскрізних та модульних тестів, але занадто мало інтеграційних тестів. Цей антипатерн призводить до численних збоїв наскрізних тестів (E2E), які могли бути виявлені швидше і простіше за допомогою набору тестів середнього рівня охоплення.

Модульні тести

Модульне тестування зосереджене на тестуванні окремих функцій або модулів коду.

- Основна мета — перевірити правильність роботи кожної частини програми ізольовано.
- Пишуться розробниками, як правило, під час або після написання функціоналу.
- Інструменти: Jest, Mocha, JUnit тощо.

Приклад: Тест функції, що додає два числа.

Юніт тест (модульний тест) перевіряє окремий юніт (наприклад, функцію, об’єкт, клас, Реакт компонент). Ви можете писати юніт тести для юніта (модуля), навіть якщо він великий і містить багато інших модулів. Щоб ефективно протестувати великий модуль, ви можете розбити його на менші, більш керовані модулі.

Необхідні ознаки юніт тесту:
1. Виконує перевірку одного юніта.
2. Локалізує проблему в межах юніта.
3. Поведінка юніта, а отже коректність юніт тесту, не залежить від сторонніх сервісів й налаштувань системи (таких як дата, база даних, стороні сайти тощо).
4. Юніт тест не тестує побічні ефекти за межами юніта, які викликав юніт.

“Модульне тестування (або юніт тестування) — це процес тестування окремих підпрограм, класів або процедур у програмі” (“Мистецтво тестування програмного забезпечення” Гленфорда Маєрса)

Приклад автоматичного юніт тесту:

```
// Функція для додавання двох чисел
function add(a, b) {  return a + b;
} // Юніт-тести для функції add()

describe('add function', () => {  // Тест на перевірку додавання двох позитивних чисел
test('adds two positive numbers', () => {  expect(add(1, 2)).toBe(3);
});

 // Тест на перевірку додавання від'ємного та позитивного чисел
test('adds a negative and a positive number', () => {  expect(add(-1, 2)).toBe(1);
 });
 // Тест на перевірку додавання двох від'ємних чисел
test('adds two negative numbers', () => {  expect(add(-1, -2)).toBe(-3);
 });
 // Тест на перевірку додавання нуля до числа
test('adds zero to a number', () => {  expect(add(5, 0)).toBe(5);
 });
});
```

Інтеграційні тести
Інтеграційні тести перевіряють взаємодію між різними модулями або компонентами системи.
- Мета — переконатися, що модулі коректно працюють разом.
- Інструменти: Postman, Supertest, TestNG.

Приклад: Тест взаємодії фронтенду та API.

Системні тести

Перевіряють всю систему в цілому, включаючи її функціональність, продуктивність і безпеку.
- Зазвичай виконуються на повністю зібраній системі.

Приклад: Перевірка роботи інтернет-магазину: від вибору товарів до оформлення замовлення.

E2E (End-to-End)
Цей підвид системного тестування перевіряє наскрізний сценарій використання системи від початку до кінця.
- Імітує реальну поведінку користувачів.
- Інструменти: Cypress, Selenium, Playwright.

Приклад: Тест реєстрації нового користувача та входу в систему.

Acceptance criteria, Acceptance test, Definition of Done
Відповідно до рекомендації Стівена Кові “починати з уявлення кінцевої мети” (Covey 2004), заздалегідь визначте, як ви оцінюватимете, чи відповідає укладений продукт вашим вимогам і вимогам ваших клієнтів. Як ви вирішуватимете, чи здійснювати остаточний платіж постачальнику? Якщо критерії приймання не будуть повністю виконані, хто буде відповідальним за виправлення, і хто їх оплачуватиме?

Приймальні тести (Acceptance tests) є системними тестами "чорного ящика". Кожен тест прийняття представляє очікуваний результат роботи системи. Замовники відповідають за перевірку коректності тестів прийняття та аналіз результатів тестування, щоб визначити, які з невдалих тестів мають найвищий пріоритет. Тести прийняття також використовуються як регресійні тести перед випуском у продакшн.
Користувацька історія не вважається завершеною, доки не пройде всі тести прийняття. Це означає, що нові тести прийняття повинні створюватися на кожній ітерації, інакше команда розробників звітуватиме про нульовий прогрес.

Смоук тестування (димне тестування) має найвищу пріоритетність (не рахуючи тести на етапі самої збірки модулів в білд, які робляться автоматично).
Саме тому смоук тестування застосовують для виявлення найгрубіших помилок, при наявності яких далі немає змісту тестувати.
Тому в умовах обмеженого часу, саме смоук тестування грає важливу роль.
Пріоритетність якраз говорить про це.
Суть смоуку не поверхневість, а найважливіші умови (вимоги).
Задача смоука виявити якнайшвидше (найекономніше) чи білд придатний до подальшого тестування.

Виходить маємо три умови смоук (димного) тестування:
1. Виявити чи білд придатний до тестування.
2. Зробити це якомога швидше, інакше щось у смоуку зайве.

3. Смоук тестування проводиться як мінімум з частотою збирання нових білдів (збірок).

Причому білд можна сприймати як білд всього додатку, або як білд окремого функціоналу (модуля). Залежно від того, що змінилося.

Якщо вас попросили посадити дерево, і після цього ви провели детальні тести листка дерева, то це не означає що саме дерево є, може посадили тільки листок) Тобто це буде порушення смоук тестування, бо, в даній аналогії, ви б мали перевірити спочатку чи є стовбур в дерева.

Я розглядаю смоук тестування як такий собі лайфхак як і класи еквівалентності. Суть в тому, щоб зробити максимально ефективне тестування за обмежений час.

Аналогія смоук (диму) взята з області електроніки. Коли тобі приносять плату на перевірку, спочатку ви дивитеся чи деталі на платі закріплені (це робить збиральник модулів автоматично), потім проводите смоук: вмикаєте в розетку плату, і дивитися чи не йде димок. Головне читайте чи на платі 220 вольт, чи 12 вольт, а то димок таки піде)
На практиці, коли ми вкручуємо лампочку, ми робимо смоук тест. (В магазинах є навіть прилади для цього).
Взагалі Steve McConnell писав, що Microsoft проводило смоук тестування для кожної нової збірки вже в 1996 році.

Саніті тестування може бути синонімом смоук тестування.
Саніті - це не про санітарію (sanitary), а про здоровий глузд (sane).
Але часто саніті застосовують як другий етап після смоук, трішки детальніший та локалізований.

Саме смоук тести мають бути автоматизовані найпершими.

Типи тестування

Регресійне тестування (Regression Testing)
- Мета: Переконатися, що нові зміни в коді не вплинули негативно на вже існуючий функціонал.
- Приклад: Повторне виконання набору тестів після додавання нової функції, щоб перевірити, чи не зламалася попередня функціональність.
- Коли застосовувати: Для підтримання стабільності системи під час активної розробки.

До регресійного тестування належить тестування знімків.
Т**естування знімків (**Snapshot Testing)
- Мета: Зберегти поточний стан системи або інтерфейсу (UI) і порівняти його з попередньо збереженою версією.

- Приклад: У фронтенд-розробці зберігається знімок (у вигляді html коду тощо) відображення React-компонента, а під час майбутніх тестів перевіряються небажані зміни.
- Коли застосовувати: Для перевірки, що зміни в UI є запланованими, а не випадковими.

Стрес-тестування (Stress Testing)
- Мета: Оцінити поведінку системи в умовах екстремального навантаження або обмежених ресурсів.
- Приклад: Моделювання одночасної роботи 1,000,000 користувачів у веб-застосунку, щоб знайти точку збою.
- Коли застосовувати: Для перевірки здатності системи витримувати пікові навантаження і відновлюватися після збоїв.

Тестування навантаження (load testing) - тип тестування продуктивності, який проводиться для оцінки поведінки компонента або системи під змінним навантаженням, зазвичай між очікуваними умовами низького, типового та максимального використання.

Тестування на проникнення (Penetration Testing)
- Мета: Виявити вразливості системи та оцінити її безпеку шляхом імітації кібернападів.
- Приклад: Тестування процесу автентифікації веб-застосунку для пошуку слабких місць.
- Коли застосовувати: Для забезпечення безпеки та запобігання зламам.

A/B тестування (A/B Testing)
- Мета: Порівняти дві версії системи або її компонента (наприклад, дизайн UI або алгоритм) для визначення ефективнішої версії на основі поведінки користувачів чи метрик.
- Приклад: Перевірка двох варіантів головної сторінки сайту для виявлення того, який збільшує кількість конверсій.
- Коли застосовувати: У маркетингу та оптимізації продуктів.

Тест план та техніки тест дизайну

Техніки тест-дизайну поділяються на три основні групи: статичні, динамічні та дослідницькі. Статичні техніки (рецензування, аналіз коду) використовуються для перевірки документації чи коду без виконання програми. Динамічні техніки включають підходи чорного ящика (еквівалентне розбиття, аналіз граничних значень) для тестування функціональності та білого ящика (покриття умов, шляхів) для перевірки внутрішньої логіки. Дослідницькі техніки, як-от ad-hoc тестування, спрямовані на пошук дефектів без чіткої стратегії.

Вибір техніки залежить від цілей тестування. Наприклад, техніки чорного ящика ефективні для перевірки функцій системи, білого ящика — для детального аналізу логіки коду, а дослідницькі підходи корисні для виявлення прихованих помилок. Комбінація цих методів дозволяє забезпечити всебічне тестування і високу якість продукту.

Тест план
Тест план — це документ, який описує:
1. Мету тестування (що потрібно перевірити).
2. Обсяг роботи (які функції, модулі чи системи тестуються).
3. Стратегію тестування (методи, підходи, середовище).
4. Ресурси (тестувальники, обладнання, програмне забезпечення).
5. Критерії прийняття (успішність тесту).
6. Ризики та їх вирішення.

Цей документ допомагає планувати тестування та координувати команду.

Тест дизайн
Тест дизайн — це процес створення тестових випадків (тест кейсів) для перевірки системи чи її частин. Основні етапи:
1. Аналіз вимог.
2. Визначення тестових сценаріїв.
3. Розробка тест кейсів.
4. Оптимізація.

Це відповідає на питання: які тести потрібні, щоб покрити всі можливі сценарії.

Клас еквівалентності (Class of equivalence)
Класи еквівалентності — це метод розбиття вхідних даних на групи (класи), які обробляються однаково. Наприклад, якщо є форма для введення віку (1–100 років), можна створити такі класи:
- Допустимі: 1–100.
- Недопустимі: <1 або >100.

Тестувати потрібно лише один представник із кожного класу.

Граничні значення (Boundary values)
Метод, що перевіряє вхідні дані на межах допустимих значень. Для віку 1–100 це:
- Мінімальні межі: 0, 1, 2.
- Максимальні межі: 99, 100, 101.

Це дозволяє знайти помилки, які часто виникають при обробці крайніх значень.

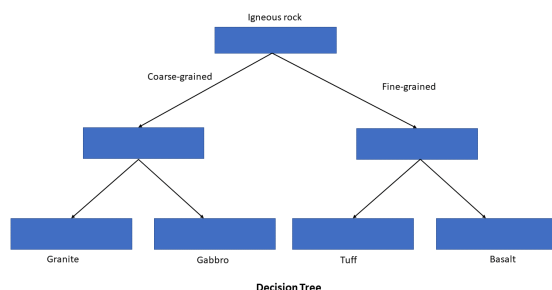
Top Down і Bottom Up decomposition
Методи інтеграційного тестування:
1. Top Down — тестування починається з високорівневих модулів. Нижчі модулі замінюються заглушками.
   - Перевага: раннє тестування основної логіки.
   - Недолік: складно реалізувати без заглушок.

2. Bottom Up — тестування починається з низькорівневих модулів. Верхні замінюються драйверами.
   - Перевага: стабільність тестування на початкових етапах.
   - Недолік: повна логіка перевіряється пізно.

Публікація тест кейсів необхідна
Це процес створення, збереження та обміну тест кейсами в системах керування тестуванням (наприклад, TestRail чи Jira).
1. Мета: Забезпечити прозорість для команди та зацікавлених сторін.
2. Включає:
   - Опис тесту (що перевіряється).
   - Очікуваний результат.
   - Реальний результат.
3. Результат: Оптимізується робота команди й підвищується якість тестування.

—

Дерево рішень — це структура, схожа на блок-схему, де кожен внутрішній вузол представляє собою "тест" на атрибут (наприклад, чи випала решка чи орел при підкиданні монети), кожна гілка відображає результат тесту, а кожен листовий вузол — це класова мітка (рішення, яке прийнято після обчислення всіх атрибутів).



Decision Tree

Дерево рішень використовується в машинному навчанні та статистиці для прийняття рішень на основі набору атрибутів. Кожен шлях від кореня дерева до листа представляє собою послідовність перевірок, що дозволяє приймати рішення чи класифікувати об'єкти. Це зручний і інтуїтивно зрозумілий метод для розв'язання задач, де потрібно приймати рішення на основі кількох змінних.

У тестуванні дерево рішень використовується для моделювання та прийняття рішень на основі різних умов, що можуть виникнути під час виконання тестів. Кожен вузол дерева представляє перевірку певної умови або атрибута, кожна гілка — це результат перевірки (наприклад, "так" чи "ні"), а кожен листовий вузол — це результат тесту чи тестове рішення.

Дерево рішень в тестуванні застосовується для автоматизації процесу вибору тестових сценаріїв на основі значень входів або попередніх результатів. Це дозволяє ефективно визначати, які тести потрібно виконати в залежності від обставин або станів системи, і може бути корисним для тестування складних логік, таких як перевірка кількох різних комбінацій входів.

Такий підхід часто використовують для моделювання тестових випадків з кількома умовами, коли кожен етап тестування залежить від попередніх результатів перевірок.

Семантичне Версіонування (SemVer)

Розглянемо формат версії X.Y.Z (МАЖОРНА.МІНОРНА.ПАТЧ). Виправлення помилок, які не впливають на API, збільшують ПАТЧ-версію. Розширення/зміни API, що сумісні з попередньою версією, збільшують МІНОРНУ версію. Ті ж зміни API, що несумісні із минулою версією, збільшують МАЖОРНУ версію.

Цю систему названо “Семантичне Версіонування”. Відповідно до неї, номери версій і спосіб їх зміни передають інформацію про базовий код і про те, що змінено від попередньої до нової версії.

Приклад: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

Регулярний вираз для SemVer сумісний з JavaScript:

```
^(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)(?:-((?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))*))?(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$
```

Автором Специфікації Семантичного Версіонування (SemVer) є Том Престон-Вернер, засновник Gravatars та співзасновник GitHub.

Альфа тестування та бета тестування — це два етапи тестування програмного забезпечення, які допомагають забезпечити його якість перед релізом.

Альфа тестування
Це початковий етап перевірки, що проводиться внутрішньо:
1. Де? У середовищі розробника або тестувальника (іноді з емуляцією умов роботи).
2. Хто тестує? Зазвичай це команда розробників або внутрішні тестувальники.
3. Мета:
   - Виявлення критичних помилок.
   - Оцінка працездатності основних функцій.
   - Перевірка інтерфейсу та логіки роботи програми.
4. Особливості:
   - Виконується в контрольованих умовах.
   - Може бути як автоматизованим, так і ручним.

Бета тестування
Це фінальний етап тестування перед офіційним релізом:
1. Де? У реальному середовищі, на пристроях користувачів.
2. Хто тестує? Кінцеві користувачі (добровольці), які погоджуються брати участь у тестуванні.
3. Мета:
   - Перевірка роботи ПЗ в реальних умовах.
   - Отримання відгуків від користувачів.
   - Виявлення рідкісних або специфічних помилок.
4. Особливості:
   - Тестування виконується реальними користувачами без контролю розробників.
   - Орієнтоване на стабільність і зручність користування.

В контексті Agile та CI/CD, альфа- та бета-тестування адаптуються під ітераційний підхід і автоматизацію, характерні для цих методологій.

Альфа тестування в Agile та CI/CD
1. Agile:
   - Альфа тестування інтегрується в кожен спринт.
   - Виконується всередині команди на завершених фічах або MVP (minimum viable product).
   - Постійне тестування дозволяє швидко ітеративно знаходити й виправляти помилки.

2. CI/CD:
   - Включає автоматичне тестування (юніт-тести, інтеграційні тести) як частину пайплайну.
   - Альфа-тести можуть бути запущені в спеціальному середовищі (наприклад, staging) перед деплоєм у production.

- Основна мета — перевірка нового коду на сумісність із існуючими функціями.

Бета тестування в Agile та CI/CD
1. Agile:
- Бета тестування відбувається після завершення кількох ітерацій, коли продукт або його частина вже готові для кінцевих користувачів.
- Зворотний зв’язок користувачів враховується як частина наступного спринту.

2. CI/CD:
- Продукт деплоїться на production, але в обмеженому масштабі (наприклад, для групи бета-тестувальників).
- Можуть використовуватися техніки канаркового релізу або blue-green deployment, щоб знизити ризики.
- Зворотний зв’язок збирається автоматизовано (аналітика, баг-трекінг).

Як це інтегрується в Agile + CI/CD пайплайн?
1. Розробка (Dev):
- Код пишеться та проходить юніт- і інтеграційне тестування (альфа стадія).

2. Staging середовище:
- Новий функціонал перевіряється на сумісність у staging-середовищі.
- Запускаються автоматизовані або мануальні альфа-тести.

3. Production (бета):
- Обмежений реліз для реальних користувачів.
- Використання A/B тестування для порівняння старих і нових функцій.
- Виявлення реальних помилок через зворотний зв’язок.

Реліз-кандидат (Release Candidate, RC) — це версія програмного забезпечення, яка йде після бета-тестування. Вона включає всі заплановані функції та виправлення помилок, виявлених на попередніх етапах, і є майже готовою до релізу. Тестування RC зазвичай виконують внутрішні команди розробників, тестувальники (QA) та окремі довірені кінцеві користувачі або партнери. Основна мета — підтвердити стабільність, відповідність вимогам і готовність до релізу.

Canary release — це стратегія розгортання нового програмного забезпечення, за якої оновлення спочатку впроваджується для невеликої групи користувачів, а після успішного тестування поступово розгортається для всіх. Цей підхід дозволяє мінімізувати ризики, пов’язані з можливими помилками чи збоями в оновленій версії.

Назва походить від практики шахтарів використовувати канарок (птахів) для виявлення небезпечного газу в шахтах. Якщо птах поводився ненормально, це слугувало сигналом до

небезпеки. Аналогічно, у "canary release" невелика група користувачів виконує роль "канарок", тестуючи стабільність нового релізу.

Стінг у своїй пісні "Canary in a Coalmine" (1980) згадує канарку в шахті як метафору для вразливості та передчуття небезпеки. Це добре ілюструє концепцію "canary release", адже тестова група також виступає як "чутливий індикатор", що сигналізує про потенційні проблеми.

Test pipeline

Описуючи Test pipeline з урахуванням використання контейнерів, тестування на різних браузерах та операційних системах, варто розглянути наступні ключові аспекти:

---

1. Test Pipeline (Концепція):
Pipeline для тестування зазвичай включає кілька етапів:
- Build Stage:
  - Збірка коду або контейнера (наприклад, за допомогою Docker).
  - Підготовка залежностей, як-от бібліотек і тестових фреймворків.

- Test Environment Setup:
  - Створення ізольованого середовища для тестування (наприклад, за допомогою Docker Compose).
  - Налаштування контейнерів для роботи з різними ОС та браузерами.
  - Завантаження тестових конфігурацій (наприклад, файли `.env`).

- Test Execution:
  - Запуск автоматизованих тестів (Unit, Integration, End-to-End).
  - Включення тестування на різних браузерах (наприклад, Chrome, Firefox, Safari).
  - Тестування на різних ОС (Linux, Windows, macOS).

- Results and Reporting:
  - Збір результатів тестування.
  - Формування звітів (HTML, Allure Reports тощо).
  - Відправлення сповіщень про результати (Slack, Email).

---

2. Використання Контейнерів:
Контейнери (наприклад, Docker) забезпечують:
- Уніфіковане середовище для тестів.

- Можливість одночасного запуску кількох інстансів з різними браузерами.
- Ізоляцію між тестами для уникнення конфліктів.

Приклад конфігурації Docker Compose для тестування на фреймворку Playwright (формат yaml):

```yaml
# docker-compose.yml

version: "3.9"

services:
  playwright:
    image: mcr.microsoft.com/playwright:focal
    container_name: playwright-tests
    working_dir: /usr/src/app
    volumes:
      - .:/usr/src/app
    entrypoint: ["/bin/bash", "-c"]
    # Запускаємо тести для всіх браузерів
    command: >
      "npx playwright test --project=chromium &&
       npx playwright test --project=firefox &&
       npx playwright test --project=webkit"
    environment:
      - CI=true
    depends_on:
      - web

  web:
    image: node:18
    container_name: web-app
    working_dir: /usr/src/app
    volumes:
      - .:/usr/src/app
    command: ["npm", "run", "start"]
    ports:
      - "3000:3000"
```

---

3. Тестування на різних браузерах:
Для тестування на браузерах можна використовувати:
- Selenium Grid:

Дозволяє запускати тести паралельно на Chrome, Firefox, Safari та інших.
- Playwright:
  Підтримує багато браузерів із вбудованою інтеграцією.

Приклад запуску тестів Playwright на JavaScript:

```
const { chromium, firefox, webkit } = require('playwright');

(async () => {
  const browsers = [chromium, firefox, webkit];
  for (const browserType of browsers) {
    const browser = await browserType.launch();
    const page = await browser.newPage();
    await page.goto('https://example.com');
    console.log(await page.title());
    await browser.close();
  }
})();
```

---

4. Тестування на різних ОС:
Тестування на різних ОС можна виконувати через:
- Docker Multi-Arch Images: для емуляції різних ОС.
- CI/CD Інструменти:
  - GitHub Actions: підтримує запуск тестів на Linux, macOS, Windows.
  - Jenkins/TeamCity: дозволяють підключати агенти для різних ОС.

---

5. Autotest Coverage (Охоплення автотестами):
Автотестове покриття забезпечує якість продукту:
- Unit Tests: Перевіряють окремі функції або модулі (70-90% покриття).
- Integration Tests: Перевіряють взаємодію між модулями (30-50%).
- E2E Tests: Перевіряють реальний користувацький флоу (10-20%).

Інструменти для аналізу покриття:
- Jest + Istanbul: для звітів покриття коду.
- SonarQube: для комплексного аналізу.

Приклад звіту покриття Jest:
```
PASS  src/app.test.js
  ✓ renders correctly (32 ms)
```

6. CI/CD Підхід:
- Автоматизація всіх тестів через CI/CD (GitLab CI, Jenkins, GitHub Actions).
- Запуск тестів паралельно для зменшення часу виконання.
- Генерація звітів і логів для швидкого аналізу результатів.

Приклад GitHub Actions для тестів (формат yaml):

```yaml
name: Test Pipeline

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]
        browser: [chrome, firefox]

    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test -- --browser ${{ matrix.browser }}
```

Цей процес забезпечує якісне тестування на всіх рівнях, різних браузерах і ОС.

Вважайте, що ціль автоматичних тестів не довести працездатність коду, а виявити помилки (баги). Це розуміння одразу показує цінність навіть мінімального покриття коду автотестами. Якщо автотест показує помилку ви можете одразу перевірити вручну чи вона дійсно є (чи тест не хибно позитивний) і одразу почати виправляти свій код, або повертати код на допрацювання. Коли всі тести пройшли, ви робити контрольну перевірку вручну і все. Тобто автотест економить час і сили більше ніж витрачено на його написання.

Пишіть негативні й позитивні тести. Наприклад, що повинно вводитись, та що не повинно. Бо, може бути таке, що у ваш input можна вводити все, тому всі позитивні тести виконаються.
Коли потрібно писати автотести:

1. Компонент використовується в багатьох місцях.
2. Код потребує частого ретесту, регресійного тестування.
3. Компонент важко тестувати під час розробки. Допоможе TDD.

Якщо під час кожної зміни коду input компонента вам потрібно робити 30 тестів вручну, то краще спочатку написати тести, а потім розробляти компонент.

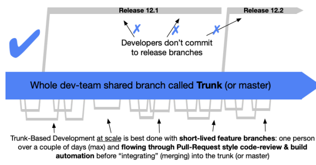
Коли не потрібно писати автотести:

1. Швидке прототипування, доказ гіпотези.
2. Маленький і простий (одноразовий) проєкт, лендінг пейдж.

Сприймайте модуль (юніт) як частину коду, яка може використовуватися незалежно. Якщо частина коду (юніт) буде використовуватися незалежно - пишіть для неї юніт тест.

Git models

Існують різні моделі Git, наприклад, GitFlow, GitHub flow, GitLab flow. Ми розмовляємо trunk-based model.



Trunk-Based Development (TBD) — це стратегія роботи з гілками в Git, де всі розробники працюють з однією основною гілкою (`trunk` або `main`). Основна ідея полягає в тому, щоб уникати довготривалих окремих гілок і регулярно інтегрувати зміни в `main`.

Основні принципи TBD:

1. Короткоживучі гілки: Зміни виконуються в маленьких гілках (feature branches), які існують лише кілька днів і швидко об'єднуються з `main`.
2. Часті інтеграції: Розробники часто пушать свої зміни в `main`, щоб зменшити ризик конфліктів.
3. Автоматизація: Інтеграція і деплой максимально автоматизовані.

Як виконується тестування в TBD:

1. Автоматизовані тести:
   - Запускаються під час кожного пушу або pull/merge request.
   - Включають unit, integration, end-to-end тести.
   - QA може створювати і підтримувати ці тести разом із розробниками.

2. Pipeline CI/CD:
   - Кожен коміт запускає pipeline, який перевіряє якість коду та працездатність нових змін.
   - Якщо тести провалюються, зміни не інтегруються в `main`.

3. Тестування на середовищі:
   - Часто використовується staging-середовище, куди автоматично деплояться зміни після успішного проходження pipeline.
   - QA проводить exploratory testing, smoke testing чи інші види мануального тестування.

4. Feature Toggles (Flags):
   - Функціональність може бути "захована" за feature toggles, щоб QA могли тестувати нові функції без впливу на користувачів.

5. Canary Testing:
   - Нові зміни поступово деплояться на невелику кількість користувачів для виявлення проблем.

Роль QA у TBD:
- Співпраця з розробниками для створення ефективних тестів.
- Аналіз ризиків і створення тест-кейсів для ключових функцій.
- Забезпечення якості продукту через автоматизацію і ретельне тестування на pre-production етапах.
- Моніторинг production, зокрема за допомогою автоматизованих інструментів.

TBD підходить для динамічних команд, які прагнуть швидкої доставки змін без втрати якості.

Refactoring та debugging

Рефакторинг (refactoring) — процес редагування програмного коду, внутрішньої структури програмного забезпечення для полегшення розуміння коду та внесення подальших правок без зміни зовнішньої поведінки самої системи.
"Щоразу, коли я роблю рефакторинг, перший крок завжди однаковий. Мені потрібно переконатися, що я маю надійний набір тестів для цієї частини коду. Перш ніж почати рефакторинг, переконайтеся, що у вас є надійний набір тестів. Рефакторинг змінює програми невеликими кроками, тому, якщо ви припуститеся помилки, легко знайти, де помилка" (Мартін Фаулер, "Рефакторінг")

Debugger вже використовували в 1960-х роках в компанії IBM.
Дебагер (Debugger) — це інструмент, який дозволяє вам виконувати крокування та відстежувати виконання програми.
В JavaScript можна використовувати ключове слово debugger, в Node.js слово inspect.
Найпоширеніша помилка, яку роблять налагоджувачі-початківці, полягає в спробі вирішити проблему шляхом внесення експериментальних змін до програми.
Не вносьте жодних змін, доки не дізнаєтеся причину помилки. Шукаємо першопричину (root cause analysis), а не лікуємо симптоми.

Налагодження (debugging) — це процес, який складається з двох частин — виявлення помилки та її усунення.
Якщо у вас виникла помилка, спочатку перегляньте програму та поясніть, як вона працює, це допоможе перевірити, чи вона розроблена так, як ви думаєте.
Локалізуйте помилку, виключивши частину коду.
Використовуйте відладчик (debugger) для проходження виконання програми.

Принципи налагодження програм:
1. Не змінюйте код, якщо не виявили суть проблеми.

2. Локалізація, Ізоляція: Відключайте частини програми (модулі), щоб виявити, який модуль спричиняє помилку. Використовуйте конструкцію try{}catch(error){} для ізоляції частин коду під час вилову помилки.

3. Метод каченяти (Rubber duck debugging) — це практика, коли програміст намагається розв'язати проблему або зрозуміти складну задачу, розмовляючи про неї з іншою людиною або навіть об'єктом, як от гумове каченя.
Метод описаний в книзі “Прагматичний програміст”. Суть методу полягає в тому, щоб викласти проблему або завдання так, ніби ви пояснюєте його іншій особі, навіть якщо ця особа фактично не відповідає. Це допомагає розібратися в проблемі, оскільки, говорячи про неї, людина змушена систематизувати свої думки і з'ясувати, що саме вона не розуміє або де саме виникла проблема. Також цей метод може допомогти виявити рішення або новий підхід до проблеми. Метод часто застосовують математики, задача яких формально описати всі інтуїтивні припущення.

4. Time traveling debugging — це метод налагодження програмного забезпечення, що дозволяє розбирати і виправляти помилки, шляхом переміщення у часі під час виконання програми. Він дозволяє розглядати стан програми на різних етапах виконання, включаючи минулі моменти, коли помилка ще не виникла.
Основні підходи до time traveling debugging включають запис та відтворення стану програми на різних етапах, використання засобів налагодження, що підтримують перехід у часі, а також використання спеціалізованих інструментів, які автоматизують цей процес.
Цей метод може бути корисним для виправлення складних помилок, які з'являються лише в певних умовах або на певних етапах виконання програми. Він дозволяє розглядати та аналізувати весь контекст, який призводить до помилки, що допомагає розуміти її причини та знайти ефективне вирішення.
Покрокове виконання коду (stepping та breakpoints) означає процес виконання програми по одній інструкції або одному рядку коду за раз. Це дозволяє розробникам спостерігати за станом програми на кожному кроці та аналізувати її поведінку, що часто використовується для налагодження.
З іншого боку, налагодження подорожуючи в часі (Time traveling debugging) — це більш просунута концепція, яка дозволяє розробникам не лише проходити код, але й рухатися назад і вперед у часі під час виконання програми.

5. Використання логування: Вбудування в програму системи логування, яка записує інформацію про роботу програми, що допомагає виявляти помилки та проблеми.

6. **Двійковий пошук багів**: Коли ви зіткнулися зі складною проблемою, подумайте про звуження масштабу проблеми до частин, які можна вирішити. Двійковий пошук по коду – це коли ви ділите код навпіл і систематично звужуєте місце розташування помилки. Такі методи, як коментування коду, використання операторів друку або ізоляція компонентів, можуть допомогти вам швидко виділити проблемну область.

Root cause (коренева причина) — це основна причина проблеми або дефекту, яка спричинила небажаний результат. Аналіз кореневих причин (Root Cause Analysis, RCA) використовується для виявлення та усунення саме тієї основної причини, щоб уникнути повторення проблеми в майбутньому. Наприклад, якщо продукт вийшов із дефектом, кореневою причиною може бути пропущений тест чи помилка в дизайні.

Affected Areas (Зачеплені області)
Affected areas – це частини програмного забезпечення чи системи, які можуть бути змінені або впливати на роботу через внесення змін. Це можуть бути:
- функції,
- модулі,
- класи,
- API,
- інтерфейси,
- бази даних,
- конфігураційні файли.

Зачеплені області визначаються під час аналізу впливу (Impact Analysis).

Impact Analysis (Аналіз впливу)
Impact Analysis – це процес оцінки потенційних наслідків змін у програмному забезпеченні. Мета цього аналізу – визначити:
1. Що саме змінюється.
   Наприклад, новий функціонал, виправлення багів чи оновлення бібліотек.
2. Які області системи будуть зачеплені.
   Визначають, які модулі, класи, тести чи інші компоненти можуть постраждати від змін.
3. Ризики змін.
   Оцінка, чи викличуть зміни нові дефекти або порушення в роботі системи.

Процес Impact Analysis

1. Огляд змін. Вивчають, що саме буде модифіковано.
2. Ідентифікація зачеплених областей.
Використовують залежності між компонентами для визначення, які області системи можуть зазнати змін.
3. Оцінка ризиків. Аналізують, як зміни вплинуть на функціональність, продуктивність, безпеку тощо.
4. Планування тестування. Визначають, які тести необхідно провести, щоб переконатися у правильності змін.

Приклад
Якщо ви змінюєте метод аутентифікації користувачів, зачеплені області можуть включати:
- логіку входу в систему,
- модулі перевірки сесій,
- API, які використовують аутентифікацію,
- мобільні додатки чи інші інтеграції.

Аналіз впливу дозволяє зрозуміти, які з цих компонентів потрібно ретельно протестувати, щоб уникнути проблем у роботі системи.

Інструменти для тестування

Інструменти для тестування — опис та класифікація:

Тест-менеджмент (управління тестами):
1. Qase
   - Хмарна платформа для управління тестами, дозволяє створювати тест-кейси, плани, виконувати звіти. Інтегрується з CI/CD та баг-трекерами.
   - Переваги: зручний UI, підтримка автоматизації через API.

2. TestRail
   - Інструмент для організації тестування та створення звітів. Підтримує інтеграцію з багатьма популярними інструментами (Jira, Selenium).
   - Переваги: багатофункціональність, зручна система звітності.

3. qTest
   - Система для управління тестуванням, підходить для Agile-процесів. Має інтеграції з DevOps-інструментами.
   - Переваги: гнучкість у масштабуванні.

Баг-трекінг:

4. Jira
   - Популярний інструмент для управління завданнями та трекінгу багів.
   - Переваги: потужні фільтри, інтеграція з різними інструментами.

5. Mantis
   - Проста у використанні баг-трекінгова система з відкритим кодом.
   - Переваги: безкоштовна, швидка у налаштуванні.

Тестування API:
6. Swagger
   - Інструмент для створення та тестування REST API, підтримує документацію.
   - Переваги: простота, зручна документація.

7. Postman
   - Інструмент для тестування API-запитів з UI та автоматизацією.
   - Переваги: багатофункціональність, підтримка колекцій тестів.

8. Apollo Explorer
   - Інструмент для тестування GraphQL API з візуалізацією запитів.
   - Переваги: інтеграція з Apollo Server, швидке налаштування.

9. GraphiQL
   - Простий інструмент для написання та тестування GraphQL API-запитів.
   - Переваги: мінімалістичний інтерфейс, відкритий код.

Тестування інтерфейсу та функцій:
10. Playwright
    - Фреймворк для end-to-end тестування веб-додатків. Підтримує багатоплатформенність.
    - Переваги: швидкість, потужний API.

11. React Testing Library
    - Інструмент для тестування React-компонентів через їх функціональність.
    - Переваги: підхід "тестування як користувач", мінімум моків.

12. Jest
    - Фреймворк для unit-тестування JavaScript та TypeScript додатків.
    - Переваги: легкість налаштування, інтеграція з багатьма бібліотеками.

13. Cucumber
    - Інструмент для BDD (Behavior-Driven Development), пише тести зрозумілою мовою (Gherkin).
    - Переваги: комунікація між технічними й нетехнічними спеціалістами.

Тестування баз даних:
14. DBeaver
- Універсальний інструмент для роботи з базами даних. Може бути корисним для перевірки даних під час тестування.
- Переваги: підтримка різних СУБД, гнучкість.
15. MySQL Workbench – Офіційний клієнт для моделювання та роботи з MySQL.
16. MongoDB Compass – Інструмент для візуального управління базами MongoDB.

Класифікація за застосуванням:
- Управління тестами: Qase, TestRail, qTest.
- Баг-трекінг: Jira, Mantis.
- API-тестування: Swagger, Postman, Apollo Explorer, GraphiQL.
- Автоматизоване тестування: Playwright, React Testing Library, Jest, Cucumber.
- Бази даних: DBeaver, MySQL Workbench, MongoDB Compass.

Слабка та сильна система типів

У мовах зі слабкою типізацією, таких як JavaScript, перевірка коректності параметрів є важливим аспектом, оскільки відсутність статичного аналізу може призвести до помилок під час виконання.

Зазначені методи допомагають мінімізувати ці ризики:

1. Юніт тести:
Юніт тестування дозволяє перевіряти коректність роботи функцій та модулів із різними наборами вхідних даних. Це не тільки підтверджує правильність логіки, але й захищає від регресій у майбутньому.

Приклад із Jest:

```
function add(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('Parameters must be numbers');
  }
  return a + b;
}

// Тестування
test('adds two numbers correctly', () => {
  expect(add(2, 3)).toBe(5);
});

test('throws an error if parameters are not numbers', () => {
```

```
  expect(() => add(2, '3')).toThrow('Parameters must be numbers');
});
```

2. Narrowing:

У JavaScript narrowing — це техніка уточнення типу змінної під час виконання програми. Наприклад, можна перевірити тип даних перед виконанням операції:

```
function processInput(input) {
  if (typeof input === 'string') {
    console.log('Input is a string:', input.toUpperCase());
  } else if (typeof input === 'number') {
    console.log('Input is a number:', input.toFixed(2));
  } else {
    console.log('Unsupported type');
  }
}

processInput('hello'); // Input is a string: HELLO
processInput(42);      // Input is a number: 42.00
processInput(true);    // Unsupported type
```

3. Type Guards:
Type guards (захисники типів) використовуються для уточнення типів складних структур. Їх можна визначити як спеціальні функції:

```
function isArray(value) {
  return Array.isArray(value);
}

function processData(data) {
  if (isArray(data)) {
    console.log('Processing array:', data);
  } else {
    console.log('Processing single value:', data);
  }
}

processData([1, 2, 3]); // Processing array: [1, 2, 3]
processData(42);        // Processing single value: 42
```

4. Бібліотеки типу zod.js:

Zod.js — це бібліотека для валідації даних і забезпечення типобезпеки. Вона дозволяє створювати схеми для валідації даних:

```
const z = require('zod');

const UserSchema = z.object({
  name: z.string(),
  age: z.number().min(18),
});

function validateUser(user) {
  try {
    const validatedUser = UserSchema.parse(user);
    console.log('Valid user:', validatedUser);
  } catch (error) {
    console.error('Validation error:', error.errors);
  }
}

validateUser({ name: 'Alice', age: 25 }); // Valid user: { name: 'Alice', age: 25 }
validateUser({ name: 'Bob', age: 15 });  // Validation error: [ ... ]
```

Ці підходи можуть комбінуватися для створення максимально надійного коду, зокрема в поєднанні з TypeScript для статичного аналізу типів.

Перевірка типів на ходу (runtime type checking) має свій мінус, оскільки споживає додаткові ресурси під час виконання програми, на відміну від статичної перевірки, яка відбувається ще на етапі компіляції. Проте вона має й свої переваги, наприклад, забезпечує гнучкість та дозволяє перевіряти динамічні дані.

Тому будьте обережні з використанням type guards, щоб не перевантажувати виконання програми й уникнути зайвих витрат ресурсів.

Zod.js — це бібліотека для валідації та перевірки типів на ходу, яка має простий і декларативний API для створення схем валідації. Вона дозволяє визначати типи та перевіряти їх в JavaScript чи TypeScript на етапі виконання програми. Хоча Zod надає зручний спосіб валідації в реальному часі, це все ж має певні накладні витрати на ресурси, подібно до інших методів runtime перевірки типів.

Тест на вірус та Теорема Баєса

Приклад використання теореми Баєса.
Нехай є захворювання X, яким може хворіти 1% населення. Також у нас є тест, який правильно виявляє захворювання у 90% випадків, але також може давати хибно позитивні результати у 5% здорових людей.
Тепер, якщо ми хочемо знати ймовірність того, що людина справді хвора, якщо тест показав позитивний результат, ми можемо скористатися теоремою Баєса:
Позначимо події:
A: людина хвора на захворювання X;
B: тест показав позитивний результат;
Позитивний результат тесту в цьому випадку означає, що людина хвора.
Знаходимо апріорну ймовірність:
P(A) = 0.01 (ймовірність того, що випадково обрана людина хвора);
Знаходимо ймовірність тесту, якщо людина справді хвора:
P(B|A) = 0.90 (правильний результат тесту для хворих);
Знаходимо ймовірність позитивного тесту, якщо людина здорова:
P(B|¬A) = 0.05 (позитивний результат тесту для здорових);
Застосовуємо формулу Баєса:
P(A|B) = (P(B|A) * P(A)) / ((P(B|A) * P(A)) + (P(B|¬A) * P(¬A)));
P(A|B) = (0.90 * 0.01) / ((0.90 * 0.01) + (0.05 * 0.99));
Частина (0.90 * 0.01) вказує на ймовірність того, що людина справді хвора і тест покаже позитивний результат.
Частина (0.05 * 0.99) вказує на ймовірність отримати позитивний результат тесту, якщо людина здорова.
Тобто (0.90 * 0.01) + (0.05 * 0.99) вказує на загальну ймовірність отримати позитивний результат тесту.
Ймовірність того, що піддослідний хворий:
P(A|B) ≈ 0.153;
Таким чином, навіть якщо тест показав позитивний результат, існує тільки близько 15.3% шансів того, що людина справді хвора. Без проведення тесту, шанс (ймовірність), що людина дійсно хвора 1%.
Позитивний тест, в нашому випадку, підвищує шанс більше ніж в 15 разів.

Тест Тюрінга

Тест Тюрінга – це метод перевірки здатності машини імітувати людський інтелект, запропонований Аланом Тюрінгом у 1950 році. В експерименті беруть участь людина-екзаменатор, машина (штучний інтелект) і людина. Всі вони спілкуються через текстовий інтерфейс. Екзаменатор ставить питання, щоб визначити, хто з двох відповідачів є машиною. Якщо екзаменатор не може впевнено визначити, де людина, а де машина, то вважається, що машина успішно пройшла тест.

Цей тест використовується для оцінки здатності штучного інтелекту взаємодіяти природно, але він часто критикується за те, що вимірює лише здатність до імітації, а не реальний інтелект або розуміння. Попри це, ідея тесту залишається важливим еталоном у дослідженнях штучного інтелекту й етичних дискусіях про межі технологій.

Тест Тюрінга вимагає опис вимог, які має виконувати середньостатистична людина.

Поведінкові антипатерни тестувальника

1. Тестувальник-Користувач
Тестувальник-Користувач відгукується про роботу розробника у суб'єктивному стилі, як це роблять звичайні користувачі, коли додаток вже випущений. Але ж не це очікує розробник від результатів тестування.

2. Тестувальник-Ледар
Тестувальник-Ледар обурюється коли знаходить баги, бо він не хоче виконувати свою роботу, а хоче, щоб код був одразу ідеальний. Йому лінь робити ретест.

3. Тестувальник-Наївний
Тестувальник, який каже розробнику, що це все елементарно. Як можна такі помилки робити?! Зазвичай він помиляється. Така поведінка може погіршити взаємини з розробниками.

4. Тестувальник-Сомельє (він же Гурман)
Тестувальник, який вважає, що його найняли в якості критика, сомельє, за його відчуття краси й витончене відчуття смаку. Такий собі гурман.

5. Тестувальник-Процесор (він же Крижане серце)
Тестувальник, який не спілкується з розробниками, а працює по принципу: "прийняв в тест, описав, повернув".

Однак спілкування з розробниками важливе для узгодження тест кейсів та для усунення факторів помилок.

Джерела

Myers, Glenford J., “The art of software testing”.

Crispin, Lisa,
“Agile testing: a practical guide for testers and agile teams”.

Jorgensen, Paul C., “Software Testing: A Craftsman’s Approach”.

Kaner, C., Bach, J., & Pettichord, B., “Lessons Learned in Software Testing: A Context-Driven Approach”.

Copeland, Lee, “A Practitioner's Guide to Software Test Design”.

Graham, Dorothy, “Foundations of Software Testing".

Whittaker, James, “Exploratory Software Testing".

Osherove, Roy, "The Art of Unit Testing".

Weinberg, Gerald M., “Perfect Software and Other Illusions About Testing".

Beck, Kent, “Test-Driven Development: By Example”.

Martin, Robert C., “Clean Code: A Handbook of Agile Software Craftsmanship”.

Martin, Robert C., “Clean Architecture: A Craftsman’s Guide to Software Structure and Design”.

Beck, Kent, and Cynthia Andres, “Extreme Programming Explained”.

Fowler, Martin, “Refactoring”.

Wiegers, Karl, “Software Requirements”.

OWASP Web Security Testing Guide

ISTQB Certified Tester AI Testing Syllabus